BOLT BERANEK AND NEWMAN INC

CONSULTING • DEVELOPMENT • RESEARCH

BBN Report No. 2932 1 March 1974

REQUIREMENTS FOR ADVANCED PROGRAMMING SYSTEMS FOR LIST PROCESSING

by

Daniel G. Bobrow

Computer Science Division
Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

This research was supported by the Advanced Research Projects Agency under ARPA Order No. 1967; Contract No. DAHC-71-C-0088

Unclassified
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Massachusetts 02138

2a. REPORT SECURITY CLASSIFICATION

Unclassified

2b. GROUP

3. REPORT TITLE

REQUIREMENTS FOR ADVANCED PROGRAMMING SYSTEMS FOR LIST PROCESSING

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Scientific

5. AUTHOR(S) *(First name, middle initial, last name)*

Daniel G. Bobrow

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
1 March 1972	33	40

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
DAHC15-71-C-0088	BBN Report No. 2339
b. PROJECT NO. ARPA ON 1967	
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. DISTRIBUTION STATEMENT

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce for sale to the general public.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
This research was sponsored by the Advanced Research Projects Agency under ARPA Order No. 1967.	

13. ABSTRACT

List processing systems should be designed to facilitate production of large programs to manipulate large complex symbolic data stores. This paper presents an overview of a number of system features which the author feels are important to improve the productivity of programmers working in such domains. A systems view is taken, rather than focusing just on language features, since algorithms not only must be coded in a language form, but debugged, modified, made efficient and run on data. Because of this genral framework, the requirements specified are applicable to the design of advanced programming systems for a wide range of applications.

DD FORM 1 NOV 65 1473 (PAGE 1)

S/N 0101-807-6811

Unclassified
Security Classification

A-31408

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
list processing						
programming languages						
design of programming languages						
interactive systems						
control structures						
data structures						
programming primitives						
semantics						
advanced programming systems						

REQUIREMENTS FOR ADVANCED PROGRAMMING SYSTEMS FOR LIST PROCESSING

by

Daniel G. Bobrow

Computer Science Division
Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

Abstract

List processing systems should be designed to facilitate production of large programs to manipulate large complex symbolic data stores. This paper presents an overview of a number of system features which the author feels are important to improve the productivity of programmers working in such domains. A systems view is taken, rather than focusing just on language features, since algorithms not only must be coded in a language form, but debugged, modified, made efficient and run on data. Because of this general framework, the requirements specified are applicable to the design of advanced programming systems for a wide range of applications.

Three aspects of programming systems are highlighted: good interactive facilities, programmable control structures, and sophisticated data communication mechanisms. Interactive features are described to facilitate program composition, entry, testing debugging, editing, optimization and packaging. Implementation of a generalized environment structure model specified would allow programming of various control regimes including multi-processes, coroutines and backtracking. Alternative methods of procedure invocation required include invocation by pattern and by monitoring condition. The need for extended data forms, storage management and extensibility are stressed, as is the duality of data retrieval and function evaluation. Syntax directed input and output of data would facilitate use of complex data stores.

TABLE OF CONTENTS

Requirements for Advanced Programming Systems for List Processing

Daniel G. Bobrow*
Bolt Beranek and Newman Inc.
Cambridge, Massachusetts 02138

1. Introduction

Programming systems are (or should be) vehicles for communication of an algorithm from a programmer to a computer in a manner which matches the needs of the programmer for this problem. As so aptly stated by Perlis in his 1964 Turing lecture:[26]

> "Programmers should never be satisfied with languages which permit them to program everything, but to program nothing of interest easily."

List processing languages are designed to make it easier to program algorithms in domains requiring manipulation of complex symbolic data structures. Examples of such problem domains are English language understanding (e.g. Winograd,[40] Bobrow)[3]; program writing programs (e.g. Green)[14]; and programming system implementation (e.g. Wegbreit).[37] This paper is a collection of my prejudices about features which should be included in a programming system to facilitate program construction in these domains. Because the general framework used is common to most difficult programming tasks, the requirements specified are applicable to the design of advanced programming systems for a wide range of applications.

Three aspects of programming systems are highlighted: good interactive facilities, programmable complex environment structures, and sophisticated data manipulation and communication

*Now at Xerox Palo Alto Research Center, Palo Alto, California

facilities. We stress the need for every system mechanism to be accessible to the programmer, following another aphorism attributed to Perlis: "One man's constant is another man's variable."

No existing system contains all of the features described below, though almost all of the ideas are implemented in some form in some current system. In those cases where it has been appropriate, I have described implementations as exemplars of requirements. The systems most influential in my thinking have been LISP 1.5,[21] on which I cut my programming teeth, BBN-LISP[5,36] for its interactive features and environment representation, PLANNER[16] (and MICRO-PLANNER)[32] for their way of invoking procedures, and backtrack control structure, FLIP[34] and SNOBOL[15] for their pattern matching facilities, and ECL[38] for its data type handling and generalized control structure primitives. These are typical, not necessarily original sources of these ideas; since this is not a survey paper, (cf. 6, 29) other references will be given only when they contain details which further explicate the ideas explored here.

I have emphasized the idea of a programming system rather than language since the programmer does not just express his algorithm, but must enter his program, test it, find bugs, modify it, etc. Because of the difficulty of this process for large complex programs, on-line interaction is mandatory. Natural modes of expressing desired actions must be available both in the algorithmic programming language and in the (somewhat different) context of direct user interaction with the system. Eventually, after getting his algorithm debugged for the test problems chosen, the programmer must often make an equivalent program which is more efficient over his total problem domain. To make his program useful to others, he must be able to isolate, specify and document its interactions within systems in which it can be embedded.

Because I feel future list processing systems must aid the programmer in his total task, I start by describing in section 2 some interactive features which facilitate construction of difficult programs. These are basically independent of programming language form, and involve the use of an intelligent agent as an intermediary between programmer and interpreter. In section 3 I discuss some necessary semantics of the programming language. This includes an (implementable) model for environmental (control and variable binding) structure, and emphasizes the rich variety of (necessary) control structures this makes possible. These include backtracking, coprocesses, and capabilities for programming within the language a general resource-sharing operating system. Section 4 describes data forms, a set of storage and access methods, and a general concept of syntax directed I/O for data.

2. Interactive Facilities

In the construction and debugging of complex programs, immediate interaction in an on-line fashion is mandatory. The usual cycle of program development has the following components:

0) Define the problem
1) Composition: Write algorithm in the chosen programming language
2) Entry: Enter machine readable form into computer
3) Testing: Run test cases
4) Debugging: Explore reasons for
   a) non-concurrence of result with expectations ("logic" bug) - often feeds back to 0
   b) unanticipated type or definition error ("structural" bug)
5) Editing: Revise program
6) Repetition: Cycle through selected portions 0-5
7) Optimization: Make a given version of a program more efficient
8) Packaging: Making a set of routines available for use by other people and programs

I shall indicate the facilities I believe most helpful for each of these components.

2.1 Composition

Programming systems become problem oriented systems when the unit forms available to the user closely match the unitary concepts in his problem domain. Addition of new procedures are not sufficient to provide appropriate language forms. Pre-processors are generally poor because debugging must be done in an interior language far removed from the external problem statement. The problem oriented language should be an extension of a powerful core language. For each problem domain the

objects, unit operations, relations and control behavior should be directly modelable in the base language augmented with new data types, new operators (or extensions of the old ones), new syntactic patterns, and new control structures, respectively. Control and data structure extension facilities will be discussed in sections 3 and 4.

Syntactic extension should be achievable by construction and modification of appropriate data forms. During the parsing of input, the parser should be able to invoke any defined procedure in the system. For example, reading a particular statement should be able to change the grammar at that time, for some defined scope. This allows somewhat incompatible sublanguages to exist within a single encompassing language - e.g. "+" may be used for a format control character in a format statement and a concatenation operator within a string matching and reconstruction statement, although in arithmetic assignment it may maintain its usual meaning. In addition to new forms, the language must allow extensions of old forms. Thus the parser must be able to use context and type information associated with variables and forms to allow generation of appropriate internal form. The parser should also be callable as a subroutine; we discuss an important application for the parser subroutine in section 4, for syntax directed data input-output.

2.2 Entry and Editing

The internal form of programs generated by the parser should be list structures with unique internal pointers for each unique external symbol (e.g. as in LISP[2] and ECL). This has several advantages. First, it can be directly executed by the interpreter without reparsing each time a line is encountered, as for example it would many times in executing a loop. Second, a list structure is a convenient input for a compiler, which can be considerably simplified when working on

this preparsed input. Third, it is a good data structure on which a programmable syntax-directed editor can work.

A good interactive editor is an important keystone in the rapid development of complex programs. The edit program in BBN-LISP serves as a model for the breadth of scope I believe is necessary. This editor is oriented toward structures, and does not allow the user to put in non-well-formed structures accidently, although partial structures can be added. It provides a large variety of commands for moving around the structure, both incrementally and by search. Searches can be specified by general structural patterns, and/or by character oriented specifications of lexical units. Various forms of on-line output allow the user to track easily where he is in the structure, even when interacting through a slow output device such as a teletype. Structural units can be deleted, inserted and moved as units. Substructures can be embedded in and extracted from larger units. Iteration of operation sequences, and conditional operations are all easily expressible. New editing operations are definable as macros in terms of older ones.

However, a major reason for the comfort and smoothness of operation in the BBN-LISP editor is the confidence the user can feel in making tentative changes in his program structure. This is achieved because the editor saves sufficient information about any changes made by an edit operation on a particular function that upon command it can UNDO that change. The feeling is very different than that achieved by saving a copy of the function definition before editing because individual changes can be undone selectively. Even after leaving the editor (which is a function called from the LISP executive) and performing other computations, the user can later reenter the editor to work on that function and UNDO changes made earlier which were unsatisfactory.

2.3 Dubugging

It is an axiom that no real program is ever written without any bugs. The bugs that occur on running a test case are of two broad kinds - "logic" bugs and "form" bugs. In the first, the programmer is trying to follow the logic of his program through a test case which is returning a wrong result. He must be able to trace parameters and values of individual functions and specific value changes within functions. This can be done by declaring specific function calls and variable bindings "sensitive"; calls to sensitive functions, and modification of sensitive variable values are monitored, and processing is suspended whenever an associated programmer-specified predicate is true. Another useful monitored condition in the system provides an interrupt (without context loss) at any function call when the programmer has pressed a special console key.

In the "break" (suspended state), the programmer must be able to examine the environment in which the condition occurred. A selective "backtrace" of the call structure (the sequence of function calls which lead to this point) and associated variable bindings must be available. In accord with our general philosophy of mechanism accessibility, this should not just be a special system print facility; any process should be able to obtain the symbolic name of the Nth preceding function, and the names and values of any variables bound in that environment. This information should be available for both regular compiled code as well as interpreted code.

When a problem is isolated, an appropriate change in a function must be made. The user should be able to call the editor while still in the break, have his changes take immediate effect and test a subcase without losing his context.

By having the editor work directly on the list structure processed by the interpreter, effects of changes are propagated immediately. For some errors, it is important (to allow continuation) for a user to have access to program substructures stored on the stack.

The second class, "structural" errors, cause interruption without programmer intervention. These include internal error conditions such as fixed or floating overflow, value-operator incongruence, or end-of-file. When an error is encountered the system should act as if the user had explicitly called a specific error handling routine, if the user had enabled such a call (e.g. say by defining an appropriately named function). If not, the system should call the break routine, described above, which preserves the state of the computation and requests user interaction from the console. Asynchronous external interrupts, e.g. a clock time-out, can be treated similarly through specially enabled interrupt handlers and the break package.

There are a class of errors in programs which can often be fixed by a programmer knowledgeable about the system but with no knowledge of the particular program. Automating such error correction, as Teitelman[35] has done in a number of cases for BBN-LISP, greatly facilitates construction of complex programs because it allows the programmer to remain thinking about his program operation at a relatively high level without having to descend into manipulation of details. Typical errors corrected by the BBN-LISP DWIM (Do What I Mean) facility include misspelled variable and function names, and syntax errors due to mistyping and mismatched parentheses.

Machine aided debugging and verification of program correctness will be an important feature to be added to list processing systems. Statements of a program's "intentions" and expected conditions at a point will become part of the program.[17] The simplest step will run these intention statements

to verify them on test cases supplied by the user. A next step will be for key test cases to be generated from the intention statements. Finally, proofs of the correctness of the program, and perhaps the program itself will be generated from the intention statements. For real problems however, I think that only the first two steps will take place in the next five years.

2.4 Testing and Repetition

A user at a console thinks of his sequence of inputs as related in terms of the program he is trying to construct and debug. However, most systems treat each interchange between the man and machine as an isolated event, and store no information from the interaction except possibly as a side effect of the computation requested. Facilities in BBN-LISP, which we describe briefly here, illustrate the power of having the user invoke system procedures through an (active) intermediary agent.

For each input to the BBN-LISP executive, the agent saves a copy of the input, the resulting value and some other information. The user can later retrieve this input by an interaction sequence number, or through search on contents of the inputs or values. He may do this, for example, to redo a test case after making some program changes. Or, he may have typed in a long expression with a slight error, and he wants to fix and reevaluate the input. Fixing is done by calling the edit facility described earlier. A set of inputs may be grouped to be done and redone as a unit. Simultaneous substitutions for a number of lexical units and/or character strings in a single command allow after-the-fact parameterization and repetition of previous input commands. Surveillance of the input by the agent provides a focusing mechanism for highlighting appropriate candidates for comparison with misspelled or mistyped units to be corrected by DWIM.

In testing programs which work on complex data structures, programs which function incorrectly may make changes in global data structure which are hard for the user to explicitly reverse. The BBN-LISP executive allows execution of generalized assignment procedures in a test mode where enough additional information is saved with an input event to allow the resulting structural changes to be undone. As with the undoing in the editor, this allows the user to try programs which might modify data structures in unexpected and not easily reversible ways, without preparing beforehand. Being forced to think about possible destructive consequences could distract the programmer from the main task at hand.

Another feature which facilitates experimentation in program interfacing is indirect procedure calls through a symbol table or transfer vector. This allows temporary use of "advice" to a function, that is a program fragment inserted between a caller and a callee; it is useful for special case handling, information gathering, and insertion of measuring probes without requiring changes in the procedure body. This facility also allows implementation of trace and break features described earlier.

2.5 Optimization and Packaging

Once a program has been written and successfully tested on a few samples of data, one often desires a version of the program which will be more efficient in some respects, e.g. time per example, temporary storage space utilized etc. Transformation of the form of the program to one more suitable for running by the computer hardware is one way of achieving some efficiencies. However, since programs are only relatively stable, compiled and interpreted code should be freely intermixable, both in terms of running and debugging. Compiling consists of making best use of information which is invariant with respect to the input data by performing all allowable computations at compile time. Only decisions which depend upon

the input data must be postponed until runtime. Compile-time activities can include transforming certain recursively written programs into equivalent iterative processes, data type and subscript bounds checking, data packing and extraction decisions, and evaluation of expressions by the compiler.[29] Such activities need not be limited only to the compiler. Program transformation under special fixed conditions is useful in other cases, e.g. incremental computation or partial evaluation[10] in which an n argument function is transformed to an n-k argument function by supplying k values.

The compiler should be able to make use of program behavior information provided by the user e.g. working set boundaries, and proportional choice of program branches. In addition, facilities must be available for the user to gather appropriate data about resource use of particular subroutines. This allows him to isolate those portions of his larger program which are critical to efficient operation of the program as a whole.

Packaging a set of programs for use by other people requires both highlighting and hiding information. Internal structure (e.g. local variable and function names) should be hidden so that, for example, use of a package doesn't surprisingly preempt the naive user's name-space. This packaging should be obtainable after the fact, so that programs can be built incrementally, as is most important for complex tasks. Preplanning the package boundaries beforehand should be unnecessary e.g. as in the BBN-LISP block-compiler. To use the block-compiler, the user specifies a group of functions previously debugged which are to be bound together. Names which are to be used external to the block are specified, and all others are surpressed in this special form of compilation.

Information to be highlighted in packaging includes interface and operating specifications. Easy association of commentary with program statements is critical for adaptation of packages for new use. Facilities which aid in documentation of programs will become more and more important, e.g. automatic flow-charters, program formatters and syntax directed documentation (c.f. Mills).[23]

3. Program Semantics

The virtual machine a user sees is determined by the primitive procedures he can call, the techniques allowable for invoking procedures and binding variables, and the richness of control structures available to express sequences of processing and modes of sharing. We will not consider here data manipulation primitives; we will assume a rich enough set to deal with the forms described in the next section, and also appropriate iteration operators. Pattern decomposition and structure building from templates are assumed as well as standard imperative statements and expressions. Because flexible control and access structures are so vital, and are usually ignored, we describe in some detail a data oriented model for control (in the spirit of Wegner[39]). We specify three primitive functions for manipulating such environmental structures which allow flexible programming of control regimes, e.g. coroutine calls and backtracking. Finally, three distinct methods of procedure invocation are described, the usual explicit call, call by pattern, and call by monitoring condition; variable binding by pairing and binding by extraction are also discussed.

3.1 A Data Structure Model of Control

In our control structure model, the primitive unit of program is an access module, e.g. a function or block, in which new nomenclature is introduced. Bindings, name-value pairs, can only be introduced at entry to an access module, i.e. as function variables or as local parameters defined at the head of a block. New modules are activated by calling or entering, and normally deactivated by exiting. A particular activation can be continued by returning to it with a value (which may be ignored).

In a single process environment, control resides in an activation of a particular module. In conventional programming languages, such a module activation has a unique caller, which in turn has a unique caller, etc; that is, the activation (calling) sequence is strictly hierarchical. We call this linear chain of activations a hierarchy in contrast to a dendrarchy, a more inclusive general control tree. We note that in a hierarchy once an activation has been exited, it disappears and can not be continued again. Reentering the module causes a new activation of the module to be created. We create a dendrarchy, a tree of activations, by providing a way to preserve an activation which has been exited, or to reference an activation from more than one successor. Each activation will have a unique caller, but may be reachable from any number of other modules.

To make our model clear, we consider the data structure required to implement it. When an access module is entered, storage is allocated for use within the module. The allocated storage we call a frame; a frame contains two major components: a basic frame and a frame extension. The basic frame is a fixed block which contains the new named items which are defined when the access module is entered. In addition, it contains a control link and an access link. The control link points to the frame of the unique calling module. The access link and bindings are used to determine the value of any variable used in the module. The formal parameters and local variables of a module may be accessed directly through the bindings component of the module frame. If a variable is not a formal parameter or local variable, it is said to be free. The value of a free variable is determined from an environment specified by the access link. Three common alternatives used for free variables are:

1) static (or lexical) scoping - as in a block structured language such as Algol. The access and control links in Algol are called the static and dynamic links respectively. The position of a variable declaration in the program text determines the free variable binding frame.

2) dynamic scoping - as in LISP. Variables are scoped according to the control flow, and the most recently defined variable of that name is assumed. The control and access links are usually identical.

3) global scoping - as in FORTRAN. A standard common area is used for free variables.

What is wanted is a system which has sufficient variability so that the user can specify the free variable access environment independent of the control environment specification. The original LISP A-list[20] provided one way of doing it; this access problem is discussed by Moses[25] in connection with the LISP funarg problem. Our frame model allows complete flexibility of access specification.

The frame extension contains anonymous temporary intermediate results of computation. At the time of a call (entry to a lower module), the caller stores in his frame extension a continuation point for the computation. For proper value checking, an expected return value type may also be stored. Since the continuation point is stored in the caller, the generalized return is simply a pointer to the frame extension of the last active frame. A point to note about a frame for an access module is that it has no pointer to any frame of a module below it; if an appropriate value (as specified by a return type) is provided, continuation in that access module can be achieved with only a pointer to the continued frame. No information stored outside this frame is necessary. Because independent returns to a frame may require distinct continuation points and temporary storage, a separate copy of the frame extension must be made for each independent successor. This is the reason for separation of the frame extension and the basic frame.

In this model for function (and block) activation, each frame is generally released upon exit of that function. Only if a frame is still referenced is it retained. Non-chained references to a frame (and to the environment structure it heads) are always made through a special data type called an environment descriptor. Only three primitive functions are needed to manipulate environments in this model. The functions: 1) environ creates an environment descriptor (ed) for a specified frame; 2) setenv changes the contents of an existing ed to point to a specified frame; 3) enveval creates a new frame with the access link specified by one ed and the control link specified by another (perhaps different) ed; it executes a specified computation in the context of that new frame.

We shall describe the environment structures for two common control regimes. Coroutines are coordinated processes which each maintain their own separate hierarchical control and access environments, with some shared base. In Figure 1, two coroutines are shown which share common access and control environment A. Note that the frame extension of A has been copied so that returns from B and Q may go to different continuation points. In Figure 2, coroutine Q is shown calling a function D with external access chain through B, but with control to return to Q. Coroutines maintain a current environment descriptor for themselves in a common data structure, and resume other coroutines through enveval.

Backtracking is a control regime in which certain environments are saved before a function return, and later restored if needed. This can be simply implemented in the model by saving an environment descriptor for that frame. As an example of its use, consider a function which returns one (selected) value from a set of computed values but can effectively return an alternative selection if the first selection was inadequate. That is, the current process can fail back to a previously specified failset point and then redo the computation with a new selection.

A sequence of different selections can lead to a stack of failset points, and successive fails can restart at each in turn. Backtracking thus provides a way of doing a depth first-search of a tree with return to previous branch points. Hewitt[18], and Golomb and Baumert[13] have discussed the use of backtracking in problem solving, and Floyd[12] discusses it as an implementation of non-deterministic programming.

An important point to note is that as we have described it, backtracking restores the control and access environment chains, but not necessarily values of shared bindings or forms of data structures which previously existed at the backtrack point. In many cases the undoing of operations to completely restore the context of a computation is what is wanted; however, control backtracking and automatic undoing of data modifications should be separably programmable. As indicated in sections 2.2 and 2.4, there are other important applications of undoing, and times when it is worthwhile to maintain careful control of which operations are reversed.

Both of the control regimes described above are important, but more important is that these regimes (and others) should be programmable. Addition of an interrupt handling facility would allow programming the equivalent of a full time-sharing system. The framework used by Prenner[27] for ECL seems adequate for this purpose. A distinguished process, called the control interpreter, is defined with two unique properties: 1) timer interrupts pass directly to it, and 2) there is a control primitive by which other processes can call for the execution of an arbitrary procedure in the environment of the control interpreter and wait for the result. The control interpreter could be made to act as a scheduler, and could also implement the Dijkstra[9] semaphore operators in a controlled environment.

With the framework of the environment structures and a control interpreter, it is straightforward to implement most

other known control structures in addition to those already shown, e.g., multiple parallel returns,[11] fork/join structures, etc., and to program others as needed. This model is developed more fully in Bobrow and Wegbreit.[7] In addition they describe a stack implementation technique which is much more efficient than the obvious heap allocation and garbage collection implementation (as used, for example, in PAL). For the usual hierarchical structure, their algorithm acts identically to the standard stack allocation scheme.

3.2 Invocation of Procedures

A procedure P is usually defined with a list of N argument names (X1, ..., XN) which have significance within the body of the procedure. An explicit procedure call P(A1, ...AN) provides N arguments to be passed to the procedure and paired as values with the N names in the definition (perhaps with data type checking and/or conversion). We call this mode of argument passing binding by pairing.

An alternative binding scheme is often attractive when working with complex symbolic structures. A procedure P is defined with a structural pattern of applicability. A single structural data unit is passed to the procedure and this data is decomposed to match the pattern. A side effect of certain of the pattern matches is to bind variables to matching substructures extracted from the input data, thus effecting what I call binding by extraction. Such pattern matching sublanguages, illustrated by SNOBOL, QA4, FLIP and PLANNER among others, are a very important facility in some list processing applications. All of those listed provide basic pattern matching facilities and more important, ways of extending the set of primitive patterns by appropriately associating a new one with a matching procedure.

As an alternative to explicit procedure call, patterns of applicability can be used implicitly to invoke procedure execution. This is the basic invoking mechanism used in PLANNER. A focus of control is a data structure; through careful indexing, likely matching procedures are tried in some retrieval order. A matching procedure is one whose applicability pattern matches the focus data. When a match is found the body of the procedure is executed. In this pattern directed invocation of procedures, a choice can be made to use one, some, or all matching procedures, with backtrack control possible to reenter the invocation sequence after successful returns. Binding is usually done by decomposition in pattern directed invocation.

The two procedure invocation methods described are immediate in their effect. Deferred procedure calls on occurrence of a specified condition are required. This monitoring function,[11] which is a generalization of the ON CONDITION of PL/I, should handle both external interrupt conditions, and changes to internal data structures. When more than one monitoring procedure is evoked by a particular environment change, a mediating priority decision maker must be invoked and must be programmable by the user.

4. Data System

The data system is the vehicle for handling all storage management and reference. It must include primitives for handling dynamically allocated data types, and named external stores of both the random access (e.g. disc files) and data stream (e.g. mag-tape, network connection) type. A rich variety of basic data types is not sufficient; general data type extension mechanisms[33] are required. Data input and output requires syntax-directed processing. The duality of selection of subparts of a data element and evaluation of a function can be broadened to good effect.

4.1 Storage Management

There are basically two types of storage to be allocated in user direct virtual memory. The first is storage automatically allocated and released on access module entry and exit. As shown in Bobrow and Wegbreit this can be done using a single stack even if multiprocesses are allowed. The second represents the storage independent of environment structures, and is allocated out of the heap, to use the Algol 68 terminology.[8] Since keeping track of storage still in use can be a substantial unwanted burden for a programmer (i.e. an item may be referenced from many different places in a data structure), automatic storage reclamation is required.[4] A requirement for garbage collection is that the system must have a well-defined, accessible base from which all storage still accessible in the system can be reached. Easily used interfaces with the allocation, pointer trace, mark, collect and relocate subroutines must be provided so that user can add his own data types. A data type extension facility which compiles and inserts appropriate code from a data description should be

provided. ECL provides a good example of how this might be conveniently done.

4.2 Dynamically Allocated Data Forms and Extensions

Any modern programming language must include a variety of basic data types. Integers and reals are required for numeric calculation, and extended precision numbers for some applications. Basic data types also include Booleans for relational values, character-strings for labels, arrays for larger fixed units of structure, and pointers (perhaps typed) to reference any data element. An important data type in list processing is the symbol, (e.g. the literal atom in LISP) which is a data item with a name, and associated internal data. It provides a mechanism for run-time symbolic interaction with a data base, i.e. a link between an external name and an internal data structure manipulable at run-time.

An important trend in use of list processing languages is the representation of information in procedural form.[18] Program-construction programs and program-modification programs (e.g. the editor described earlier) will be a standard part of the library of users of these languages. These require that procedures be a manipulable basic data type in the system.

No list of data types can be complete, and therefore a data type extension facility is mandatory. Extension facilities provide several mechanisms for defining a new type; as a homogeneous array of previously defined type with subitems selectable by index; as a heterogeneous structure with subitems selectable by name; as pointers (references) to other data types; and as a data type which is the union of previously defined types, (i.e. is one of several types, to be determined at runtime). A data type description must be used (say by

compilation of appropriate code) for:

1. Construction of objects of this type: Optimization of storage efficiency can be obtained by intelligent compilation of data procedures to pack components in minimal space.
2. Selection of components from the item (for compound objects)
3. Assignment to items of this type, and to components (for compound objects)
4. Garbage collection, as described earlier.

Certain operations are defined over each data type. In some languages (e.g. SIMULA)[19] the user can extend a data type to a new subclass such that all operations on the original type are still applicable in addition to any new operations; e.g. a LISP-like cell could be extended to have n-additional data words, but still allow all the usual list operations such as car, cdr, cons. Procedures represented as lists are another important data subclass because all list manipulation operations are applicable for modifying these structures.

All properties of a specific data type should be under possible control by the programmer by allowing him a simple way to interpose his own procedure for any of the standard assumed procedures associated with the object. This may sometimes be costly, requiring runtime interpretation instead of open compiled code.

4.3 External Data Forms and Syntax-Directed Data I/O

List processing systems work within operating systems which provide permanent storage facilities, and methods for communicating with the user. Care must be taken to ensure that these facilities are easily accessible and

manipulable from within the list processing system. We distinguish three basic categories of external data form: random access stores, sequential data streams, and graphical (two dimensional) I/O.

Random access stores should be treatable as extensions of the main memory space. The only differences are that an additional argument (the store-name) must be somehow provided for the access procedures; and the costs of operations to an external store may be different than to main virtual memory.

Sequential data streams are one dimensional and must use implicit grammatical relationships to indicate structural links. In order to process an input stream, a language needs more than simple format statements. Letting the user program his own input routine with primitives, while necessary, does not facilitate enough the construction of complex data bases. A sublanguage designed for the task is necessary, and this is where one can extend the use of the general parsing routine usually associated only with program input. The parsing tables can be modified to accept the external data form, and the code generators replaced by data constructors.

Constructing data streams from structured data is also necessary. If the data language input is defined by an unambiguous context free grammar, a generator for the language (from the internal data) should be able to be computed from the parsing tables. Even if they must be defined separately, facilities for syntax-directed data input and output are important for future list processing systems.

Graphical input-output has an important subcase which appears in a number of systems. This is the generation of formatted listings of programs. Printing programs in a "pretty" form which reflects their internal structure is nice for file oriented systems, and imperative where programs are

modified or constructed within the system, and are only available in the internal representation.

General graphic output languages have been developed, and such facilities will be useful in problem domains where the two dimensions of the picture (perhaps with projections from higher dimensions) aid in understanding complex data structures. Graphic input is considerably less developed, and requires further work in parsing of 2 dimensional patterns[1] to be really useful. However, utilization of such a facility is important when the problem domain has a natural expression in the form of line drawings (e.g. architectural design) or where spatial position has meaning (e.g. symbolic mathematical manipulation).

4.4 Duality of Function Evaluation and Data Selection

Selection of a data item from a structure requires specification of the structure and the item name or index. For a multidimensional structure, several indices may be provided. From these inputs a location of the required item is usually determined, and the data extracted. Alternatively, one can think of selection as looking up a datum stored associated with an n-tuple of names.

In evaluating a function we usually think of performing a sequence of computations based on the inputs to obtain a value. Obviously evaluation and selection are just two ends of a single spectrum, reflecting a trade-off in space and speed. Both ends of the spectrum should be expanded toward the middle. Selection of some data items should be allowed to invoke a user defined function, both generic for the data type and specific for an instance. Such functional indirection would

allow, for instance, automatic extraction of current information functionally dependent on other items, or propagating information on storage of new data.

At the other end of the spectrum, the concept of "memo functions" introduced by Michie[22] allows function evaluation to degenerate to data retrieval in common cases. POP-2[10] doublets, in which an "updater" function (which stores values for a specified set of arguments), is paired with an ordinary valued function, allows the usual assignment and selection operations for data to apply to functions. An early use of a similar technique was made in Samuels[31] rote learning of the evaluation of checker positions; in some cases he could look up a previously computed look-ahead value of a position. This enabled him to extend the effective depth of his look-ahead significantly. In general, noting the duality of function evaluation and data retrieval allows the flexibility of space-time tradeoffs so necessary in solving complex problems in list processing systems. Gedanken is based in part on this principle of duality of function evaluation and data selection.

4.5 Data Access Methods

In addition to data extraction, it is important in many domains to be able to retrieve data items by content. Given a pattern, all data items matching that pattern should be obtainable from a data base. Basic indexing and sorting facilities are necessary to implement such retrieval packages. Associative links from one, or combinations of key items, implemented by hash addressing techniques will allow access for certain classes of data; this is done in PLANNER, among others. QA4 sorts all input expressions through a discrimination tree, with equivalence over different variable names. A pattern there defines a set of nodes in the tree below which are all matching data items; these leaves of the tree are the retrieval nodes.

Not all data in the data base need have the same scope of applicability. Associated with each piece of data (perhaps implicitly) needs to be context information which defines the scope of its validity. If the contexts are tree structured, several incompatible data bases with different amounts of shared structure can exist (e.g. QA4[30]).

Complex network structures with many explicit links are common in symbolic processing applications, especially those concerned with natural language processing. There are a number of cases in which additional information is required about a particular data structure node, but an explicit link from the data item to this information is forbidden - e.g. a large read-only data store, and temporary tags marking recent processing on the node. Another example is associating a property SIMPLIFIED with a particular substructure of a symbolic expression without interfering with its mathematical form. A technique recently added to BBN-LISP allows such implicit links. It utilizes association arrays; a location in such an array is computed on the basis of the internal address of the data item. In this hash-link location is stored the original address and the associated information. Care must be taken to disambiguate collisions in this hash array, and to provide appropriate mechanisms to garbage collect the implicit link when the data item is no longer referenced.

5. Conclusion

Programs to solve complex problems often evolve over a long time period. A good environment will allow sets of users to build up a collection of consistent tools which aid in solving their problems. Built on a core system which is relatively easily transferable to new hardware, it can provide a machine independent base for a wide range of activities.

A list processing system can't be all things to all people all the time. However, with a flexible set of features which span the problem space, extension facilities, a large library of useful routines, and good documentation, a system should help a programmer to stand on the shoulders of his predecessors, not his toes.

6. Acknowledgements

The writing of this paper was supported by the Advanced Research Projects Agency. The author would like to thank Bob Balzer, Peter Deutsch, Gerald Sussman, Warren Teitelman, Robert Thomas and Ben Wegbreit for discussions concerning various aspects of the material contained here; and Madeline Morin for her help in preparation of this manuscript.

REFERENCES

1. Anderson, R.H. Syntax directed recognition of hand printed two dimensional mathematics. Ph.D. Thesis Harvard University, Jan. 1968.

2. Berkeley, E.C. and Bobrow, D.G. (eds) The programming language LISP; its operation and applications. MIT Press, Cambridge, 1966.

3. Bobrow, D.G. Natural language input for a computer problem solving system. Ph.D. Thesis MIT 1964, In Minsky [23]

4. Bobrow, D.G. Storage management in LISP. Proc. IFIP Conf. on Symbol Manipulation Languages North Holland, Amsterdam 1967

5. Bobrow, D.G. and Murphy, D.L. Structure of a LISP system using two-level storage. Comm ACM 10,3 (March 1967) 155-159

6. Bobrow, D.G. and Raphael, B. A comparison of list-processing computer languages. Comm ACM 7,4 (April 1964)

7. Bobrow, D.G. and Wegbreit, B. A stack implementation of retention in coordinating sequential processes. BBN Report No. 2334, February 1972 (submitted to CACM)

8. Branquant, P., Lewi, J., Sintzoff, M., and Wodon, P.L. "The composition of semantics in Algol 68. Comm ACM 14, 11 (Nov. 1971) 697-708

9. Dijkstra, E.W. Cooperating sequential processes. In Genuys (Ed.) Programming Languages. Academic Press 1967.

10. Burstall, R.M., Collins, J.S. and Popplestone, R.J. Programming in POP-2 Edinburgh University Press, Edinburgh 1971

11. Fischer, D. Control structures for programming languages. Ph.D. Thesis Carnegie-Mellon University 1970.

12. Floyd, R.W. Nondeterministic Algorithms. JACM 14, 4 Oct. 1967.

13. Golomb, S.W. and Baumert, L.D. Backtrack programming. JACM 12, 4 (Oct. 1965) 516-524.

14. Green, C.C., Theorem proving by resolution as a basis for question-answering systems. Machine Intelligence 4 American Elsevier Publishing Company, Inc. New York 1969

15. Griswold, R.E., Poage, J.R. and Polonsky, L.P. The SNOBOL 4 programming language Prentice-Hall Englewood, N.J. 1971

16. Hewitt, C. PLANNER: a language for manipulating models and proving theorems in a robot. Proc IJCAI Washington, D.C. 1969

17. Hewitt, C. Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot. Ph.D. Thesis MIT Feb. 1971

18. Hewitt, C. Procedural embedding of knowledge in PLANNER Proc Second IJCAI London 1971

19. Ichbiah, J.D. and Morse, S.P. General concepts of the SIMULA 67 programming language Companie Internationale pour le Informatique DR. SA. 69. 132 ND Paris Sept. 1971

20. McCarthy, J. Recursive functions of symbolic expressions. Comm ACM 3 1960 184-95

21. McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P. and Levin, M.I. LISP 1.5 programming manual MIT Press Cambridge, Mass. 1962

22. Michie, D. Memo functions: a language feature with rote-learning properties. Proc IFIP Edinburgh 1968

23. Mills, H. Syntax directed documentation for PL360. Comm ACM 13, 4 1970 216-222.

24. Minsky, M.L. (ed.) Semantic information processing MIT Press, Cambridge, Mass. 1968

25. Moses, J. The function of function in LISP ACM SIGPLAN Notices June 1969

26. Perlis, A.J. The synthesis of algorithmic systems, J. ACM 14, (Jan. 1967) 1-9

27. Prenner, C. Multi-path control structures for programming languages. Ph.D. Thesis, Harvard University, June 1972.

28. Prenner, C., Spitzer, J., Wegbreit, B. An implementation of backtracking for programming languages. Submitted to ACM 72.

29. Raphael, B., Bobrow, D.G., Fein L., and Young, J.W., A brief survey of computer languages for symbolic and algebraic manipulation Proc IFIP Conf on Symbol Manipulation Languages North Holland, Amsterdam 1967

30. Rulifson, J.F., Waldinger, R.J., Dirksen, J.A. QA4, a language for writing problem-solving programs Proc IFIP Congress TA-2 P111-115 1968

31. Samuels, A.L. Some studies in machine learning using the game of checkers IBM J of Res and Dev 3,3 (1959)

32. Sussman, G.J. and Winograd, T. Micro-planner reference manual. AI Memo 203, Project MAC MIT Cambridge, Mass. (1970)

33. Standish, T.A. A data definition facility for programming languages Ph.D. Thesis Carnegie Institute of Technology, Pittsburgh, Pennsylvania May 1970

34. Teitelman, W. Design and implementation of FLIP, a LISP format directed list processor BBN Report AFCRL-67-0514 July 1967

35. Teitelman, W. Toward a programming laboratory Proc IJCAI Washington, D.C. (1969) 1-8

36. Teitelman, W., Bobrow, D.G., Hartley, A.K. and Murphy, D.L. BBN-LISP, TENEX reference manual Bolt Beranek and Newman July 1971

37. Wegbreit, B., Studies in extensible programming languages Ph.D. Thesis Harvard University 1970 (available as ESD-TR-70-297)

38. Wegbreit, B., The ECL programming system Proc FJCC 197 p. 253-262

39. Wegner, P. Information structure models Proc SIGPLAN Symposium on data structures in programming languages SIGPLAN Notices 6, (Feb 1971) pp. 1-54

40. Winograd, T. Procedures as a representation for data in a computer program for understanding natural language Ph.D. Thesis MIT 1970 Project MAC TR-84 MIT Cambridge, Mass. Feb. 1971